Università degli Studi di Bologna
Facoltà di Ingegneria

Progettazione di Applicazioni Web T

Esercitazione 4
JDBC e
Progettazione persistenza “forza bruta”

Agenda

- **Esercizio guidato completo per farci trovare pronti alla prova d'esame**

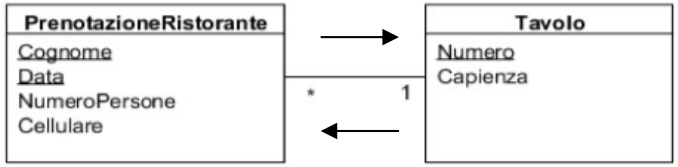
  Progettazione, implementazione e gestione della persistenza basata su metodologia «forza bruta» a partire da una realtà di interesse descritta mediante uno schema UML

- Passi principali:
  - Dall'UML ai JavaBean, dall'UML alle tabelle DB mediate progettazione logica
  - Mapping delle relazioni e dei vincoli
  - Implementazione delle query
  - Tricky elements

«Prenotazione Ristorante»

N.B. Con un piccolo abuso di notazione, nei diagrammi UML la sottolineatura di un attributo non indica la staticità di tale attributo bensì il suo vincolo di univocita alla E/R.

Partendo dalla realtà illustrata nel **diagramma UML** di seguito riportato, si fornisca una soluzione alla gestione della persistenza basata su metodologia **Forza Bruta** in grado di "mappare" il modello di dominio rappresentato dai **JavaBean** del diagramma UML con le corrispondenti **tabelle relazionali derivata dalla progettazione logica** del diagramma stesso.



Si consideri inoltre la presenza del vincolo: "*Uno stesso tavolo può essere prenotato più volte solo se in date diverse*".

Nel dettaglio, dopo aver creato da applicazione Java lo schema della tabella nel proprio schema nel database **TW_STUD** di **DB2** (esplicitando tutti i vincoli derivati dal diagramma UML) e implementato **JavaBean** e metodi necessari per la realizzazione delle **operazioni CRUD**, si richiede la definizione del metodo principale di richiesta prenotazione `boolean RichiestaPrenotazione(String Cognome, Date data, Int numeroPersone, String Cellulare)`. Tale metodo, mediante l'uso del metodo di supporto `String NumeroTavolo DisponibilitaTavolo(Date data, Int numeroPersone)`, verifica la disponibilità di almeno un tavolo di `capienza >= numeroPersone` per la data richiesta e, in caso di esito positivo, restituisce il codice numerico (`NumeroTavolo`) di uno di questi. Tale codice è usato dal metodo `RichiestaPrenotazione` per procedere all'inserimento persistente della prenotazione nel DB e alla restituzione del valore di verità `true` attestante l'accettazione della prenotazione. In caso di esito negativo di disponibilità tavolo invece, il metodo `DisponibilitaTavolo` restituisce `null`, mentre il metodo principale `RichiestaPrenotazione` restituisce direttamente il valore di verità `false` attestante il rifiuto della prenotazione.

Si richiede quindi di realizzare una classe di prova in grado di:

- inserire diversi tavoli e diverse prenotazioni nelle tabelle corrispondenti;
- utilizzare correttamente i metodi `RichiestaPrenotazione` e `DisponibilitaTavolo` per verificare la disponibilità o meno di un tavolo rispetto a una determinata richiesta (si contempli sia il caso di risposta positiva che negativa);
- produrre una stampa completa, opportunamente formattata, delle prenotazioni (complete di numero tavolo) presenti nel DB **prima e dopo** l'inserimento al punto precedente sul file **Prenotazione.txt**.

Dall'UML ai JavaBean e alle tabelle DB

- La prima cosa da fare è trasformare il diagramma UML nelle corrispondenti classi Java e tabelle DB (derivate dalla progettazione logica applicata all'UML)

- Ogni classe del diagramma corrisponde normalmente ad una classe java
- Ad ogni proprietà della classe UML corrisponde normalmente una proprietà della classe Java

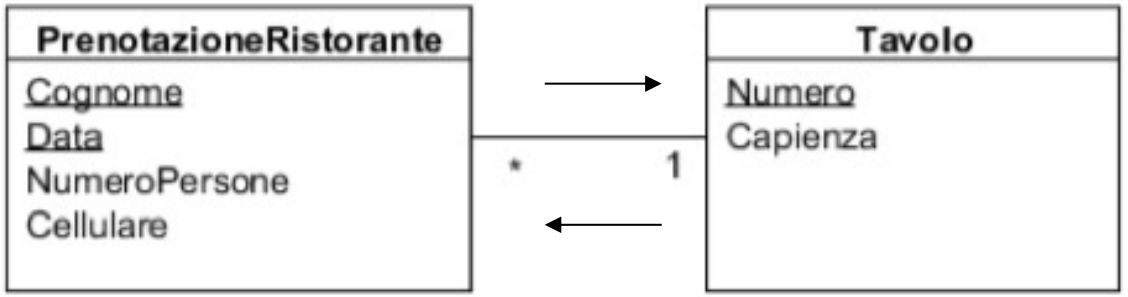
Dunque, basta contare le classi e le proprietà UML per sapere quante classi e proprietà Java modellare?
No, come vedremo non proprio è così, non ci sono mai regole preconfezionate ma tutto dipende dal contesto dell'applicazione… ☹

Mapping delle relazioni

- La seconda cosa che può influire sul numero delle classi e sul numero di attributi in Java sono le relazioni tra le classi UML:
  - **Relazione 1-N/N-1**: non comportano classi java aggiuntive, ma solo l'aggiunta di attributi; la cardinalità **N** è mappata con una collezione di oggetti lato java, mentre la cardinalità **1** con un semplice id intero
  - **Relazione N-N**: comporta la creazione di una tabella aggiuntiva nel DB, detta tabella di mapping
    - La tabella di mapping potrebbe avere una sua corrispondente classe java (si parla in questo casi di «materializzazione» degli ID), oppure no (si gestisce esplicitamente la relazione N-N rappresentata nel diagramma UML).
    - La scelta dipende anche da eventuali vincoli esplicitati nel testo dell'esercizio!
- Il numero di attributi nelle classi java dipende anche dalla eventuale richiesta nel testo di utilizzo di «ID surrogati»
  - Sono attributi solitamente di tipo intero che fungono da **identificatori univoci** degli **oggetti JavaBean** e da **primary key (PK)** nelle tabelle corrispondenti
- **! Anche se la traccia non lo richiede, per completezza presentiamo una soluzione basata su id surrogati (ipotesi extra traccia)**

Il caso concreto del nostro esercizio

- Come sarà il mapping di questo diagramma?



  - Due classi Java, una per classe UML
  - Tutti gli attributi esplicitati nel diagramma
  - Un id surrogato (int) per classe
  - Una Collezione di oggetti PrenotazioneRistorante nella classe Tavolo
  - Un campo contente la chiave materializzata (l'ID tavolo) della tabella Tavolo nella casse PrenotazioneRistorante

```
public class PrenotazioneRistorante {

    private int idPrenotazione;
    private String cognomePrenotazione;
    private Date dataPrenotazione;
    private int numeroPersonePrenotazione;
    private String cellularePrenotazione;
    private int idTavoloPrenotazione;
```

```
public class Tavolo {

    private int idTavolo;
    private String numeroTavolo;
    private int capienzaTavolo;
    private Set<PrenotazioneRistorante> prenotazioniTavolo;
```

Mapping dei vincoli (1)

- Il mapping dei vincoli viene fatto nelle classi java che si occupano dell'interfacciamento con il DB, dette «**`Repository`**»
  - C'è una classe Repository per ogni JavaBean
    - Queste classi implementano tutti i **metodi CRUD**
  - In più ci sono ulteriori due classi strettamente relative al DB fisico
    - Una per la gestione della connessione al DB denominata **`DataSource()`**
    - Una per la gestione personalizzata delle eccezioni denominata **`PersistenceException()`**

- Vincoli esplicitati nell'UML:
  - La coppia Cognome-Data in PrenotazioneRistorante è univoca (UNIQUE)
  - Il Numero tavolo in Tavolo è univoco (UNIQUE)
- Vincoli impliciti o espressi nel testo, tipo:

Si consideri inoltre la presenza del vincolo: "*Uno stesso tavolo può essere prenotato più volte solo se in date diverse*".

- ! Per date diverse si intende «giorni diversi»

Mapping dei vincoli (2)

- Vediamo come si implementano i vincoli individuati…
  - L'implementazione dei vincoli viene demandata al metodo della classe "Repository" che si occupa della creazione della tabella, ovvero alla query "Create Table"

```
// create table
private static String create =
    "CREATE " +
        "TABLE " + TABLE +" ( " +
            ID + " INT NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1 INCREMENT BY 1), " +
            COGNOME + " VARCHAR(20) NOT NULL, " +
            DATA + " DATE NOT NULL, " +
            NUMEROPERSONE + " INT, " +
            CELLULARE + " VARCHAR(10), " +
            TAVOLO + " INT NOT NULL REFERENCES tavolo, " +
            "PRIMARY KEY ("+ID+"), "+
            "CONSTRAINT pt_PrenotazioneID UNIQUE ("+COGNOME+","+DATA+"), " +
            "CONSTRAINT pr_PranotazioneTavoloID UNIQUE ("+DATA+","+TAVOLO+") " +
        ")"
    ;
```

! La definizione completa dell'id surrogato **ID** la vediamo tra qualche minuto…

Implementazione query

- Le query richieste dall'esercizio vengono implementate dai seguenti metodi:
  - `boolean RichiestaPrenotazione(String cognome, Date data, int numeroPersone, String cellulare)`
  - `String DisponibilitaTavolo(Date data, int numeroPersone)`
- Il primo metodo, tramite l'utilizzo del secondo, verifica la disponibilità di un tavolo (ovvero, **`capienza >= numero di persone`**); se è disponibile un tavolo, il secondo metodo restituisce il suo codice numerico, altrimenti null;
- Se un tavolo è disponibile, il primo metodo usa il numero di tavolo restituito per inserire persistentemente nel DB la prenotazione

Query: DisponibilitàTavolo (1)

- Il metodo deve restituire il numero di un tavolo, qualora ci fosse, con capienza >= al numero di persone richiesto ad una certa data
- Questo risultato può essere ottenuto in due modi differenti:
  1. Effettuando due query distinte, una che seleziona tutti i tavoli e l'altra tutte le prenotazioni per la data richiesta, per poi fare un controllo incrociato delle due collezioni con Java
     - Metodo complicato, laborioso e molto poco efficiente
  2. Oppure con un'unica query innestata
     - Metodo pulito, efficiente ed elegante

- La query viene implementata dal metodo dedicato nella classe «`TavoloRepository()`»

Query: DisponibilitàTavolo (2)

```
private static String DisponibilitàTavolo(Date data, int persone)
{
    return tr.availableTable(data, persone);
}
```

```
public String availableTable(Date data, int persone)
{
    String result=null;
    Connection connection = null;
    PreparedStatement statement = null;
    try{
        connection = this.dataSource.getConnection();
        statement = connection.prepareStatement(read_available_table);
        statement.setInt(1, persone);
        statement.setDate(2, data);
        ResultSet rs = statement.executeQuery();
        if(rs.next())
        {
            result = rs.getString(NUMERO);
        }
        else
            result = null;
        return result;
    }catch (SQLException e) {
        throw new PersistenceException(e.getMessage());
    }
    finally {
        try {
            if (statement != null)
                statement.close();
            if (connection!= null){
                connection.close();
                connection = null;
            }
        }
        catch (SQLException e) {
            throw new PersistenceException(e.getMessage());
        }
    }
}
```

Query: DisponibilitàTavolo (3)

- Di seguito la query innestata "*read_available_table":*

```
static String read_available_table =
        "SELECT "+ NUMERO +
            " FROM " + TABLE + " " +
            "WHERE " + "capienza" + " >= ? AND "+ID+" NOT IN ( SELECT idTavolo FROM prenotazione WHERE data = ?)";
```

- L'inner query seleziona tutti gli ID dei tavoli prenotati nella data richiesta
- L'outer query seleziona tutti I tavoli con capienza >= al numero di persone con ID diverso da tutti quelli restituiti dall'inner query

Query: RichiestaPrenotazione (1)

- Questa query prende il codice del tavolo restituito dal primo metodo e va a fare un'operazione di insert sulla tabella delle prenotazioni del ristorante

```
private static boolean RichiestaPrenotazione(String cognome, Date data, int persone, String cellulare)
        throws PersistenceException {
    String tableavailable = DisponibilitaTavolo(data, persone);
    if (tableavailable == null)
        return false;
    int numTavolo = tr.getIdFromNumber(tableavailable);
    PrenotazioneRistorante prr = new PrenotazioneRistorante();
    prr.setCellularePrenotazione(cellulare);
    prr.setCognomePrenotazione(cognome);
    prr.setDataPrenotazione(data);
    prr.setIdTavoloPrenotazione(numTavolo);
    prr.setNumeroPersonePrenotazione(persone);
    pr.persist(prr);
    return true;
}
```

- Il metodo restituisce TRUE se l'operazione è andata a buon fine, FALSE altrimenti

Query: RichiestaPrenotazione (2)

```
public void persist(PrenotazioneRistorante pr) throws PersistenceException{
    Connection connection = null;
    PreparedStatement statement = null;

        try {
            connection = this.dataSource.getConnection();
            statement = connection.prepareStatement(insert);
            statement.setString(1, pr.getCognomePrenotazione()+"");
            statement.setDate(2, pr.getDataPrenotazione());
            statement.setInt(3, pr.getNumeroPersonePrenotazione());
            statement.setString(4, pr.getCellularePrenotazione());
            statement.setInt(5, pr.getIdTavoloPrenotazione());
            statement.executeUpdate();

            statement = connection.prepareStatement(check_query);
            statement.setString(1, pr.getCognomePrenotazione());
            statement.setDate(2, pr.getDataPrenotazione());
            ResultSet rs = statement.executeQuery();
            rs.next();
            int idprenotazione = rs.getInt(1);
            pr.setIdPrenotazione(idprenotazione);
        }
        catch (SQLException e) {
            throw new PersistenceException(e.getMessage());
        }
        finally {
            try {
                if (statement != null)
                    statement.close();
                if (connection!= null){
                    connection.close();
                    connection = null;
                }
            }
            catch (SQLException e) {
                throw new PersistenceException(e.getMessage());
            }
        }
}
```

Tricky Elements

- In questo esercizio c'è un elemento più o meno esplicito…
- il metodo che rende persistente una prenotazione, non accetta tra i parametri l'ID surrogato della prenotazione presente nel rispettivo oggetto **`PrenotazioneRistorante`**
  - Ovviamente non è possibile aggiungere il parametro ID alla signature del metodo perché in questo modo non verrebbe rispettata la specifica del testo… occorre trovare un altro modo…

Tricky Elements: ID auto-incrementale

- Come possiamo inserire una nuova tupla senza preoccuparci di esplicitarne il valore?
  - Attraverso l'uso di ID auto-incrementali
- Come si implementa un ID auto-incrementale?
  - È un ID generato automaticamente dal DB all'inserimento di una nuova

```
// create table
private static String create =
    "CREATE " +
        "TABLE " + TABLE +" ( " +
            ID + " INT NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1 INCREMENT BY 1), " +
            COGNOME + " VARCHAR(20) NOT NULL, " +
            DATA + " DATE NOT NULL, " +
            NUMEROPERSONE + " INT, " +
            CELLULARE + " VARCHAR(10), " +
            TAVOLO + " INT NOT NULL REFERENCES tavolo, " +
            "PRIMARY KEY ("+ID+"), "+
            "CONSTRAINT pt_PrenotazioneID UNIQUE ("+COGNOME+","+DATA+"), " +
            "CONSTRAINT pr_PranotazioneTavoloID UNIQUE ("+DATA+","+TAVOLO+") " +
        ") "
    ;
```

! Una volta inserita la tupla nella tabella, occorre allineare il valore dell'ID generato con quello del corrispondente oggetto JavaBean